

Keywords

higher category theory, concurrency, message-passing, types, Curry-Howard

ABSTRACT

We present an approach to modeling computational calculi using higher category theory. Specifically we present a fully abstract semantics for the π -calculus. The interpretation is consistent with Curry-Howard, interpreting terms as typed morphisms, while simultaneously providing an explicit interpretation of the rewrite rules of standard operational presentations as 2-morphisms. One of the key contributions, inspired by catalysis in chemical reactions, is a method of restricting the application of 2-morphisms interpreting rewrites to specific contexts.

Higher category models of the pi-calculus

Michael Stay
Google
stay@google.com

L.G. Meredith
Biosimilarity, LLC
lgreg.meredith@biosimilarity.com

1. INTRODUCTION

One of the major distinctions in programming language semantics has been the division between denotational and operational semantics. In the former computations are interpreted as mathematical objects which—more often than not—completely unfold the computational dynamics, and are thus infinitary in form. In the latter computations are interpreted in terms of rewrite rules operating on finite syntactic structure. Historically, categorical semantics for programming languages, even variations such as games semantics [6] which capture much more of the intensional structure of computations, are distinctly denotational in flavor [11]. Meanwhile, operational semantics continues to dominate in the presentation of calculi underlying programming languages used in practice [5] [3] [13].

Motivated, in part, by the desire to make a closer connection between theory and practice, many efforts in the programming language semantics, and in concurrency theory communities have begun to investigate more direct categorical interpretations of operational semantics. This paper finds its place in this latter context, providing a fully abstract interpretation of the π -calculus in terms of a higher categorical model of its operational semantics. In particular, while it remains faithful to a Curry-Howard orientation, modeling terms as typed morphisms, it models the computational dynamics of the calculus, its rewrite rules, as 2-morphisms. One of the goals has been to provide a modular semantics to address a range of features and modeling options typically associated with the π -calculus. For example, a significant bifurcation

occurs in the treatment of names with Milner’s original calculus hiding all internal structure of names [10], while the ρ -calculus variant provides a reflective version in which names are the codes of processes [8]. The semantics presented here is capable of providing a categorical interpretation of both variants.

Of particular interest to theoreticians and implementers, the semantics shines light on a key difference between the categorical and computational machinery it interprets. The latter is intrinsically lazy in the sense that all contexts where rewrites can apply must be explicitly spelled out (cf the context rules in section 2.1.4), while the former is intrinsically eager¹; in fact, one of the contributions of the paper is the delineation of an explicit control mechanism to prevent unwanted rewrites that would otherwise create an insurmountable divergence between the two formalisms.

1.0.1 Related work

This paper draws inspiration from [15] and [2], but also seeks a more direct account of what works in modern day operational semantics. In his seminal paper [9] Milner provided the template still used today for specifying computational calculi, presenting the π -calculus in terms of a freely generated algebra quotiented by a structural equivalence relation that is then subject to some rewrite rules. This constitutes the modern view of structured operational semantics [12]. In the latter part of his research Milner focused on finding a satisfying relationship between a categorical presentation of the rewrite rules and the notion of bisimulation [4]. While this work did not explicitly employ higher categorical techniques, it spawned a variety of 2-categorical investigations designed to capture and recast bisimulation equivalences in terms of 2-morphisms [14]. Hirschowitz has developed an even more ambitious program of categorifying the whole of the operational semantics framework from the presentation of higher order syntax (or terms with binding constructors like π -calculus or λ -calculus), to rewrite

¹like the mythical hydra, chop off one 1-morphism and a 2-morphism takes its place ;-)

rules [1].

The present work is primarily focused on providing a direct account of the π -calculus. The modularity of the semantics arises from wanting to give a clean design and clear shape to the present account, rather than an attempt to provide a framework for interpreting a number of computational calculi. The fact that the techniques do apply to a number of calculi was a side effect of this process. Moreover, our particular reconciliation of operational laziness with categorical eagerness introduces an explicit resource sensitivity, which we have not seen before in the theoretical literature, yet is remarkably similar to resource constraints in actual implementations of concurrent and distributed computations.

1.0.2 Organization of the rest of the paper

In the remainder of the paper we present the core fragment of the calculus we model followed by a manifest of the categorical equipment needed to faithfully model it. Then we give the semantics function an sketch a proof that the interpretation is fully abstract.

2. THE CALCULUS

One notable feature of the π -calculus is its ability to succinctly and faithfully model a number of phenomena of concurrent and distributed computing. Competition for resources amongst autonomously executing processes is a case in point. The expression

$$x?(y) \Rightarrow P \mid x!(u) \mid x?(v) \Rightarrow Q$$

is made by composing three processes, two of which, $x?(y) \Rightarrow P$ and $x?(v) \Rightarrow Q$ are seeking input from channel x before they launch their respective continuations, P and/or Q ; while the third, $x!(u)$, is supplying output on that same said channel. Only one of the input-guarded processes will win, receiving u and binding it to the input variable, y , or respectively, v in the body of the corresponding continuation – while the loser remains in the input-guarded state awaiting input along channel x . The calculus is equinamious, treating both outcomes as equally likely, and in this regard is unlike its sequential counterpart, the λ -calculus, in that it is not *confluent*. There is no guarantee that the different branches of computation must eventually converge. Note that just adding a *new*-scope around the expression

$$(\text{new } x)(x?(y) \Rightarrow P \mid x!(u) \mid x?(v) \Rightarrow Q)$$

ensures that the competition is for a local resource, hidden from any external observer.

2.1 Our running process calculus

2.1.1 Syntax

$P ::= 0$	stopped process
$ x?(y_1, \dots, y_n) \Rightarrow P$	input
$ x!(y_1, \dots, y_n)$	output
$ (\text{new } x)P$	new channel
$ P \mid Q$	parallel

Due to space limitations we do not treat replication, $!P$.

2.1.2 Free and bound names

$$\begin{aligned} \mathcal{FN}(0) &:= \emptyset \\ \mathcal{FN}(x?(y_1, \dots, y_n) \Rightarrow P) &:= \{x\} \cup (\mathcal{FN}(P) \setminus \{y_1, \dots, y_n\}) \\ \mathcal{FN}(x!(y_1, \dots, y_n)) &:= \{x, y_1, \dots, y_n\} \\ \mathcal{FN}((\text{new } x)P) &:= \mathcal{FN}(P) \setminus \{x\} \\ \mathcal{FN}(P \mid Q) &:= \mathcal{FN}(P) \cup \mathcal{FN}(Q) \end{aligned}$$

An occurrence of x in a process P is *bound* if it is not free. The set of names occurring in a process (bound or free) is denoted by $\mathcal{N}(P)$.

2.1.3 Structural congruence

The *structural congruence* of processes, noted \equiv , is the least congruence containing α -equivalence, \equiv_α , making $(P, |, 0)$ into commutative monoids and satisfying

$$(\text{new } x)(\text{new } x)P \equiv (\text{new } x)P$$

$$(\text{new } x)(\text{new } y)P \equiv (\text{new } y)(\text{new } x)P$$

$$((\text{new } x)P) \mid Q \equiv (\text{new } x)(P \mid Q)$$

2.1.4 Operational Semantics

$$\frac{| \vec{y}| = |\vec{z}|}{x?(y) \Rightarrow P \mid x!(z) \rightarrow P\{\vec{z}/\vec{y}\}} \quad (\text{COMM})$$

In addition, we have the following context rules:

$$\frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \quad (\text{PAR})$$

$$\frac{P \rightarrow P'}{(\text{new } x)P \rightarrow (\text{new } x)P'} \quad (\text{NEW})$$

$$\frac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q} \quad (\text{EQUIV})$$

2.1.5 Bisimulation

DEFINITION 2.1.1. An observation relation, \downarrow is the smallest relation satisfying the rules below.

$$\begin{array}{ll}
 \overline{x!(\vec{y}) \downarrow x} & \text{(OUT-BARB)} \\
 \overline{P \downarrow x \text{ or } Q \downarrow x} & \text{(PAR-BARB)} \\
 \overline{P \downarrow x, x \neq u} & \text{(NEW-BARB)} \\
 (\text{new } u)P \downarrow x
 \end{array}$$

Notice that $x?(y) \Rightarrow P$ has no barb. Indeed, in π -calculus as well as other asynchronous calculi, an observer has no direct means to detect if a sent message has been received or not.

DEFINITION 2.1.2. An barbed bisimulation, is a symmetric binary relation \mathcal{S} between agents such that $P \mathcal{S} Q$ implies:

1. If $P \rightarrow P'$ then $Q \rightarrow Q'$ and $P' \mathcal{S} Q'$.
2. If $P \downarrow x$, then $Q \downarrow x$.

P is barbed bisimilar to Q , written $P \approx Q$, if $P \mathcal{S} Q$ for some barbed bisimulation \mathcal{S} .

3. CATEGORICAL MACHINERY

We take our models in 2-categories with an underlying symmetric monoidal closed category; the 2-categories Cat (categories, functors, and natural transformations) and Rel (sets, relations, and implications) are examples. We denote the monoidal unit object by I , the tensor product by \otimes , the n th tensor power of an object X by $X^{\otimes n}$, and the internal hom by a lollipop \multimap .

4. THE INTERPRETATION

Given the abstract syntax of a term calculus like that in section 2.1.1, we introduce an object in our 2-category for each parameter of the calculus. We introduce 1-morphisms for each term constructor, 2-morphisms for each reduction relation, and equations for structural equivalence; we also add 1-morphisms to mark contexts in which reductions may occur.

The π -calculus is parametric in a set of names and a set of processes, so we have objects \mathcal{N} and \mathcal{P} . Since names can be reused in the π -calculus, we also add 1-morphisms and equations to make \mathcal{N} be a cocommutative comonoid. We denote comultiplication by $\Delta: \mathcal{N} \rightarrow \mathcal{N} \otimes \mathcal{N}$ and counit by $\delta: \mathcal{N} \rightarrow I$. If the tensor product is the cartesian product, I is the terminal

object, \mathcal{N} is a comonoid in a unique way, and Δ and δ are duplication and deletion, respectively.

In the π -calculus, all reductions occur at the topmost context, so we have one unary morphism from \mathcal{P} to \mathcal{P} . There are some benefits to constructing the top context marker out of the existing binary morphism $|: \mathcal{P} \otimes \mathcal{P} \rightarrow \mathcal{P}$ and a unary morphism $\text{COMM}: I \rightarrow \mathcal{P}$; we'll talk about some of the benefits in the conclusion.

The theory of the π -calculus is the free symmetric monoidal closed 2-category on

- objects \mathcal{N} for names and \mathcal{P} for processes,

$$\begin{array}{ccc}
 & \downarrow & \downarrow \\
 \mathcal{N} & & \mathcal{P}
 \end{array}$$

- 1-morphisms $\Delta: \mathcal{N} \rightarrow \mathcal{N} \otimes \mathcal{N}$ and $\delta: \mathcal{N} \rightarrow I$,

$$\begin{array}{ccc}
 & \downarrow & \downarrow \\
 \mathcal{N} & \swarrow & \searrow \\
 & \mathcal{N} & \mathcal{N}
 \end{array}$$

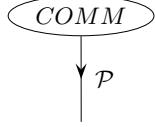
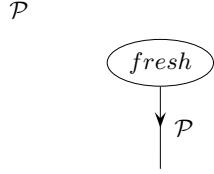
- 1-morphisms $|: \mathcal{P} \otimes \mathcal{P} \rightarrow \mathcal{P}$ and $0: I \rightarrow \mathcal{P}$,

$$\begin{array}{ccc}
 & \downarrow & \downarrow \\
 \mathcal{P} & \multimap & \mathcal{P} \\
 & \downarrow & \downarrow \\
 \text{l} & & 0 \\
 & \downarrow & \downarrow \\
 & \mathcal{P} & \mathcal{P}
 \end{array}$$

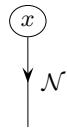
- 1-morphism $?_n: \mathcal{N} \otimes (\mathcal{N}^{\otimes n} \multimap \mathcal{P}) \rightarrow \mathcal{P}$ and $!_n: \mathcal{N} \otimes \mathcal{N}^{\otimes n} \rightarrow \mathcal{P}$ for each natural number $n \geq 0$,

$$\begin{array}{ccc}
 & \downarrow & \downarrow \\
 \mathcal{N} & \multimap & \mathcal{N}^{\otimes n} \multimap \mathcal{P} \\
 & \downarrow & \downarrow \\
 ?_n & & \mathcal{P} \\
 & \downarrow & \downarrow \\
 & \mathcal{N} & \mathcal{N}^{\otimes n} \\
 & \downarrow & \downarrow \\
 !_n & & \mathcal{P}
 \end{array}$$

- 1-morphisms $\text{fresh}: I \rightarrow \mathcal{P}$ and $\text{COMM}: I \rightarrow$

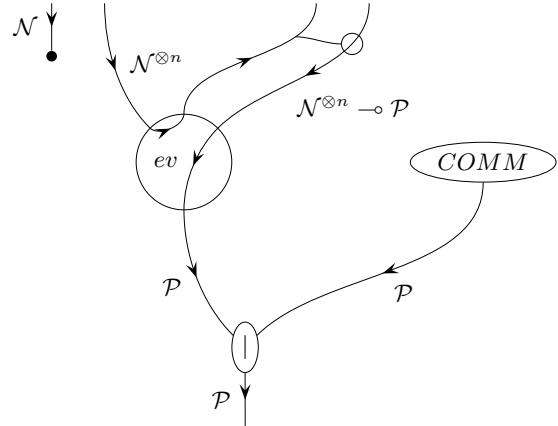


(Note that the equations governing **new** in section 2.1.3 are satisfied up to tensoring with a scalar due to the naturality of the unitors and braiding in the symmetric monoidal 2-category.) For convenience we write $\llbracket x \rrbracket$ for



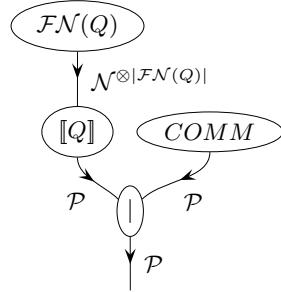
which picks out x from \mathcal{N} .

- equations making $(\mathcal{P}, |, 0)$ into a commutative monoid,
- equations making (N, Δ, δ) into a cocommutative comonoid, and
- a 2-morphism $comm_n$ encoding the COMM rule for each natural number $n \geq 0$.

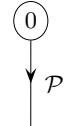


4.1 Semantics

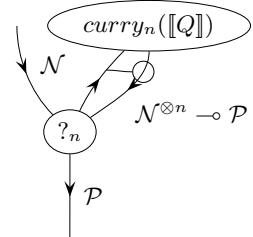
$$\llbracket Q \rrbracket_{top} :=$$



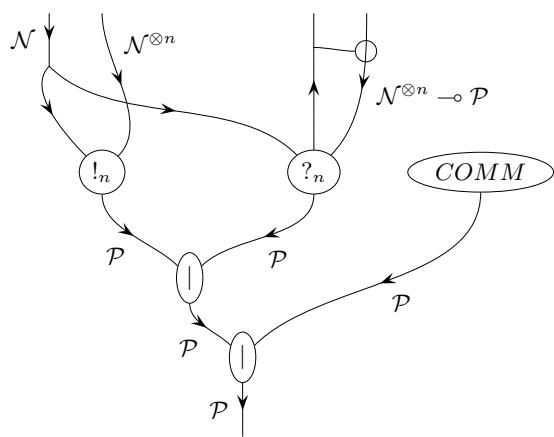
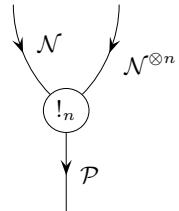
$$\llbracket 0 \rrbracket :=$$



$$\llbracket x?(y_1, \dots, y_n) \Rightarrow Q \rrbracket :=$$

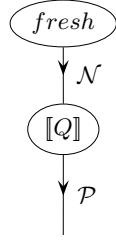


$$\llbracket x!(y_1, \dots, y_n) \rrbracket :=$$

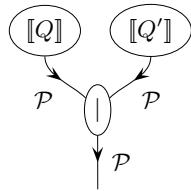


$$comm_n \Downarrow$$

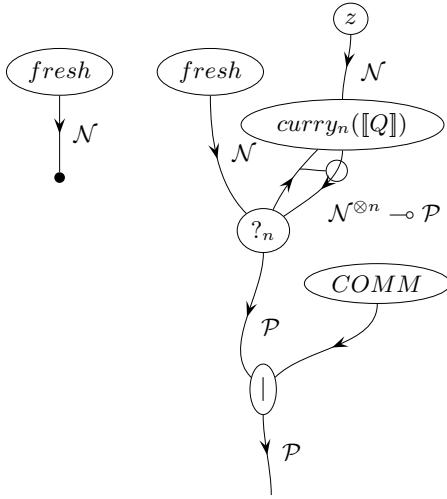
$\llbracket (\text{new } x)Q \rrbracket :=$



$\llbracket Q \mid Q' \rrbracket :=$



For example, $\llbracket (\text{new } y)(\text{new } x)x?(y_1, \dots, y_n) \Rightarrow Q \rrbracket_{top}$ where z is free in Q is



4.1.1 Bisimulation again

In this setting we can provide a direct interpretation of observation and bisimulation. Roughly speaking, $\llbracket P \rrbracket$ reduces to $\llbracket Q \rrbracket$ just when we can apply the 2-morphism $comm_n$ to the former to produce the latter. Since all non-trivial 2-morphisms are generated by $comm_n$, single step reductions $\llbracket P \rrbracket \rightarrow \llbracket Q \rrbracket$ correspond precisely to the decomposition of a $\llbracket P \rrbracket$ in terms of a ‘‘context’’ functor C , such that $\llbracket P \rrbracket = C[\text{src}(comm_n)]$, and $\llbracket Q \rrbracket = C[\text{trgt}(comm_n)]$. More generally, the interpretation of a term context $\llbracket K \rrbracket$ is a functor from one hom category to another: the functor takes an appropriate morphism f to fill the hole and returns a new morphism $\llbracket K \rrbracket(f)$; similarly, it takes a 2-morphism $\alpha: f \Rightarrow f'$ between appropriate morphisms and whiskers and/or

tensors α with identity 2-morphisms to produce a new 2-morphism $\llbracket K \rrbracket(\alpha)$.

DEFINITION 4.1.1. $\llbracket P \rrbracket \Rightarrow^{\text{comm}} \llbracket P' \rrbracket$ iff there is a 2-cell, $F: \llbracket P \rrbracket \rightarrow \llbracket P' \rrbracket$ generated by exactly one top level occurrence of $comm_n$ and horizontal and vertical composition of identity 2-morphisms.

LEMMA 4.1.2 (REDUCTION). $P \rightarrow P' \iff \llbracket P \rrbracket \Rightarrow^{\text{comm}} \llbracket P' \rrbracket$

Proof: by construction. The only subtlety here is that there be only one $COMM$ map to ensure only 1 component of P reduces, but this is just what the definition ensures. \square

Likewise, we can transport the notion of observability to the categorical setting as a relation, \Downarrow , between 1-morphisms (not necessarily in the same hom-category). More precisely, \Downarrow is the smallest relation satisfying

- $\llbracket x!(y_1, \dots, y_n) \rrbracket \Downarrow \llbracket x \rrbracket$
- $\llbracket P \rrbracket \Downarrow \llbracket x \rrbracket$ or $\llbracket Q \rrbracket \Downarrow \llbracket x \rrbracket$ implies $\llbracket P \rrbracket \mid \llbracket Q \rrbracket \Downarrow \llbracket x \rrbracket$
- $\llbracket P \rrbracket \Downarrow \llbracket x \rrbracket$, $x \neq u$ implies $\llbracket (\text{new } u)P \rrbracket \Downarrow \llbracket x \rrbracket$

LEMMA 4.1.3 (OBSERVATION). $P \Downarrow x \iff \llbracket P \rrbracket \Downarrow \llbracket x \rrbracket$

Proof: by construction. \square

Taken together these two notions provide an immediate lifting of the syntactic notion of bisimulation to a corresponding semantic notion, which we write, $\dot{\approx}$.

DEFINITION 4.1.4. $\llbracket P \rrbracket \dot{\approx} \llbracket Q \rrbracket$ iff

1. If $\llbracket P \rrbracket \Rightarrow^{\text{comm}} \llbracket P' \rrbracket$ then $\llbracket Q \rrbracket \Rightarrow^{\text{comm}} \llbracket Q' \rrbracket$ and $\llbracket P' \rrbracket \dot{\approx} \llbracket Q' \rrbracket$.
2. If $\llbracket P \rrbracket \Downarrow \llbracket x \rrbracket$, then $\llbracket Q \rrbracket \Downarrow \llbracket x \rrbracket$.

4.1.2 Full abstraction and contextual congruence

THEOREM 4.1.5 (FULL ABSTRACTION). $P \dot{\approx} Q \iff \llbracket P \rrbracket \approx \llbracket Q \rrbracket$

Proof: this follows from lemmas 4.1.2 and 4.1.3. \square

Typically, bisimulation is too rigid. Contextual congruence allows for appropriate notion of equivalence in the presence of substitutions.

DEFINITION 4.1.6 (CONTEXTUAL CONGRUENCE).
 $P \simeq Q$ iff $C[P] \approx C[Q]$ for all C .

We need the corresponding notion

DEFINITION 4.1.7 (CONTEXTUAL CONGRUENCE).
 $\llbracket P \rrbracket \dot{\simeq} \llbracket Q \rrbracket$ iff $\llbracket C \rrbracket(\llbracket P \rrbracket) \dot{\approx} \llbracket C \rrbracket(\llbracket Q \rrbracket)$ for all C .

where $\llbracket C \rrbracket$ is the functor on hom categories mentioned above. We can immediately verify that

$$\llbracket C \rrbracket(\llbracket P \rrbracket) = \llbracket C[P] \rrbracket$$

We require

$$P \simeq Q \iff \llbracket P \rrbracket \dot{\simeq} \llbracket Q \rrbracket$$

But this follows directly

$$\begin{aligned} C[P] \approx C[Q] \\ \iff (\text{bisimilarity result}) \\ \llbracket C[P] \rrbracket \approx \llbracket C[Q] \rrbracket \\ \iff (\text{definition of } \llbracket C \rrbracket) \\ \llbracket C \rrbracket(\llbracket P \rrbracket) \approx \llbracket C \rrbracket(\llbracket Q \rrbracket) \end{aligned}$$

5. CONCLUSIONS AND FUTURE WORK

We presented a fully abstract higher categorical semantics for the π -calculus. Our semantics can be seen as a natural extension of Curry-Howard in the categorical setting: if terms are taken to be 1-morphisms, then rewrites between terms should be 2-morphisms. Such an approach is natural from another perspective in that it makes comparison with operational semantics considerably simpler, at least conceptually. To that end, we have already applied the approach to models of other milestone computational calculi, such as the lazy λ -calculus, with some initial success and hope to report on that in subsequent papers.

Perhaps more importantly, establishing connections like this between two different computational frameworks should allow for transport of other key conceptual tools. Here, we were able to transport a version bisimulation to the categorical setting in a simple and straightforward manner. It would be quite interesting to be able to transport categorical notions of typing back to the process setting. For example, Mellies and Zeilberger's refinement types [7], expressed as functors, suggest an intriguing approach to a more categorical account of behavioral types.

Finally, the use of $COMM$ to control $comm_n$ based rewrites is strongly reminiscent of the distinction between logical, or virtual concurrency such as may be found in a threads package or operating system process abstraction, versus actual hardware resources. Allowing more than one $COMM$ resource provides, on the one hand a very natural account of so-called true concurrency semantics, and on the other the means to reason about these very practical situations which we hope to investigate in future work.

Acknowledgments. We would like to acknowledge Jamie Vicary for some early conversations about a higher-category-based approach; Tom Hirschowitz for asking some thoughtful and stimulating questions about earlier versions of this work; and Marius Buliga for some initial conversations which prompted us to reconsider enzymatic-style solutions.

6. REFERENCES

- [1] *Cartesian closed 2-categories and permutation equivalence in higher-order rewriting*, Logical Methods in Computer Science **9(3:10)2013**, 1–22.
- [2] Barne P. Hilken, *Towards a proof theory of rewriting: The simply typed 2lambda-calculus*, Theor. Comput. Sci. **170** (1996), no. 1-2, 407–444.
- [3] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler, *Featherweight java: A minimal core calculus for java and GJ*, Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999. (Brent Hailpern, Linda M. Northrop, and A. Michael Berman, eds.), ACM, 1999, pp. 132–146.
- [4] James J. Leifer and Robin Milner, *Deriving bisimulation congruences for reactive systems*, CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings (Catuscia Palamidessi, ed.), Lecture Notes in Computer Science, vol. 1877, Springer, 2000, pp. 243–258.
- [5] Sergio Maffeis, John C. Mitchell, and Ankur Taly, *An operational semantics for javascript*, Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings (G. Ramalingam, ed.), Lecture Notes in Computer Science, vol. 5356, Springer, 2008, pp. 307–325.
- [6] Guy McCusker, *Games and full abstraction for FPC*, Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New

Brunswick, New Jersey, USA, July 27-30, 1996, IEEE Computer Society, 1996, pp. 174–183.

[7] Paul-André Melliès and Noam Zeilberger, *Functors are type refinement systems*, Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015 (Sriram K. Rajamani and David Walker, eds.), ACM, 2015, pp. 3–16.

[8] L. Gregory Meredith and Matthias Radestock, *A reflective higher-order calculus.*, Electr. Notes Theor. Comput. Sci. **141** (2005), no. 5, 49–67.

[9] Robin Milner, *Functions as processes*, Mathematical Structures in Computer Science **2** (1992), no. 2, 119–141.

[10] ———, *The polyadic π -calculus: A tutorial*, Logic and Algebra of Specification Springer-Verlag (1993).

[11] Eugenio Moggi, *Notions of computation and monads*, Inf. Comput. **93** (1991), no. 1, 55–92.

[12] Gordon D. Plotkin, *The origins of structural operational semantics*, Journal of Logic and Algebraic Programming, 2004, pp. 60–61.

[13] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi, *Python: The full monty*, Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (New York, NY, USA), OOPSLA '13, ACM, 2013, pp. 217–232.

[14] Vladimiro Sassone and Pawe L Sobociński, *Deriving bisimulation congruences: A 2-categorical approach*, In FOSSACS 03, volume 2620 of LNCS, Springer, 2002, pp. 409–424.

[15] R. A. G. Seely, *Modelling computations: A 2-categorical framework*, Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987, IEEE Computer Society, 1987, pp. 65–71.